Spring 2020

A Mixed Reality System for Learning Data Structures

Cullen C. Drissell
Bard College, cd6586@bard.edu

A Mixed Reality System For Learning Data Structures

A Senior Project submitted to The Division
of Science, Mathematics, and Computing
of
Bard College

by
Cullen Drissell

Annandale-on-Hudson, New York
May, 2020

Abstract

This project seeks to create an educational tool that makes learning and conceptualizing data structures easier for students studying computer science. The final product uses a camera-projector system as well as multiple computer vision techniques to detect physical wooden tokens on a surface. These tokens are then processed and represented as nodes in a graph. The program applies algorithms to this graph and shows visualizations to help the user, or users, better understand the interaction between data structures and algorithms.

Contents

Dedication

I dedicate this project to my family and close friends who have all helped me through this process in their own ways.

Acknowledgments

I would like to thank my advisor, Keith O'Hara, for helping and inspiring me throughout my four years here at Bard. You have been helpful beyond words in both teaching and preparing me for entering into the world of computing and programming. Without your aid, I would not have been able to achieve as much as I have. Thank you.

I would also like to thank my family, Don, Tracy, Lexi, and Brogan. Without you guys, I would not have ended up in New York studying computer science. You have always been very supportive of whatever I am pursuing.

Another big thank you to my girlfriend, Gianna, who has been with me through my whole experience here at Bard and who has helped me in more ways than she knows. Thank you for always being there for me.

I would also like to thank all of my good friends who have made all of my experiences at Bard truly awesome. I could not have asked for a better crew to spend the last four years with.

1. Introduction

1.1 Problem

The study of computer science contains many topics that can be difficult to understand and conceptualize without aid. The formulation of data structures and their interconnection with algorithms are integral parts to both computer science and programming. The visual representations of data structures can be very helpful when learning about their connection to code and algorithms. Interaction with code is often mediated through a screen and keyboard. Words on a screen and a computer's abstraction of these words within the hardware can be daunting and confusing for people who are learning how to code.

This project seeks to create a tool for people learning about code, algorithms, and data structures. The final product uses a camera and small, circular wooden tokens to create a mixed reality system that allows a user to physically manipulate and visualize a graph data structure in real time. I will implement a minimum spanning tree algorithm on the graph to show how algorithms interact with data structures. A visual representation of this interaction, as well as the graph itself, will be displayed on the user's computer screen. As they interact with the physical tokens, the visualizations on the screen will adapt to the movements of the nodes in the graph. This will allow users to better understand the structure of the graph data structure and conceptualize how it is affected by the minimum spanning tree algorithm. This project could be expanded to include other algorithms and data structures to create a complete tool for effectively teaching these central topics in computer science.

1.2 Related Work

1.2.1 Dynamicland

I found inspiration for my senior project from the non-profit research group known as Dynamicland based in Oakland, California. In 2018, they opened a community space that houses a vast network of projector-camera systems that are used to create a hands-on and dynamic way of interacting with computers. Paper, pens, scissors, glue, and other materials are used as tools for computing. The main goal of Dynamicland is to create a communal computer that offers people a space to interact face-to-face while learning, playing, and working. “In Dynamicland, computational media isn't hidden away in isolated virtual worlds. It's real stuff that everyone can see and get their hands on... And everyone gets their hands on everything! You walk by, you see what someone is making, you play with it and trade ideas, you sit down and work on it together. This happens constantly” [1]. Dynamicland’s commitment to creating physical and dynamic ways of interacting with both computers and people are central to my Senior Project.

Figure 1.2.1. Shows an example of a workspace at Dynamicland [2].

1.2.2 Python Tutor

Python Tutor, a web-based programming tool, is another point of inspiration for my project. The website helps users visualize programming functions and data structures. The user inputs a function into the browser and has the ability to step through its execution. The tool offers the ability to view run-time states of data structures, stack frames, variables, as well as heap object contents and pointers [3]. The data structure visualization aspect of PythonTutor is what I am using as a point of inspiration for my senior project.

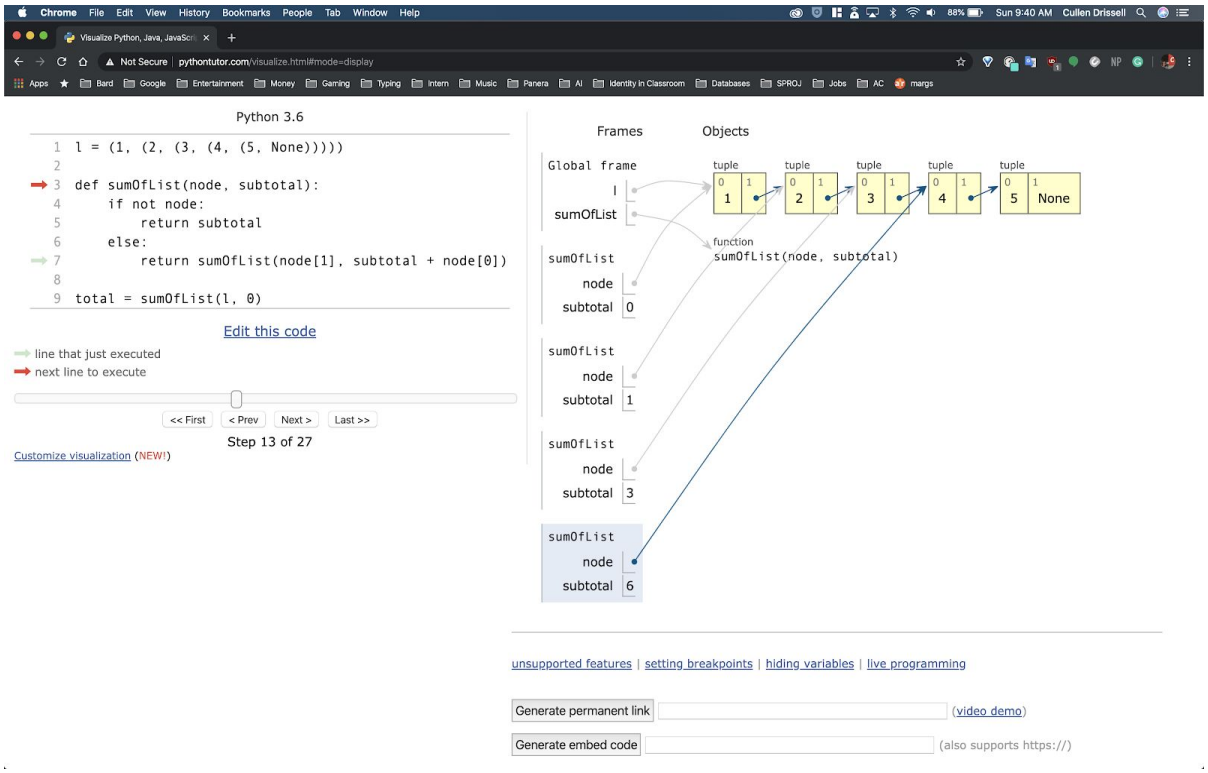


Figure 1.2.2. A screenshot from pythontutor.com

2. Approach/Methods

2.1 OpenCV

OpenCV is an open source computer vision library created in 1999 at Intel by Gary Bradski. It is a large library of algorithms mostly used for computer vision. As an open source project, OpenCV is expanded and added to daily. It supports many programming languages such as C++, Python, and Java. It is also available on most popular operating systems today such as Windows, Mac OS X, Linux, Android, and iOS. The OpenCV tools I use for this project include text detection and hough circle transform [4].

2.2 Projector-Camera Graph Program

I began working on a projector-camera system that would allow a user or a group of users to physically manipulate a data structure in real time. I chose to use the graph data structure as my data structure of choice for this program because I began to use small wooden circle tokens for circle detection which resembled nodes in a graph.

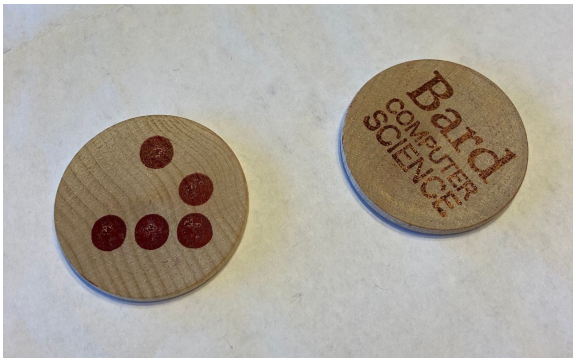


Figure 2.2.1. The wood tokens I used for circle detection

The program I wrote called *projector_circle_detect.py* uses **OpenCV** for hough circle transform, **imutils** for video manipulation, and **numpy** for matrix operations for image windows. The main function of the program is to read in an image from a webcam, detect circles in the image which will act as the nodes in a graph, and then ask the user to input values for these nodes to be stored in the graph. After this, the user can connect the nodes in the graph and eventually find a path between two nodes.

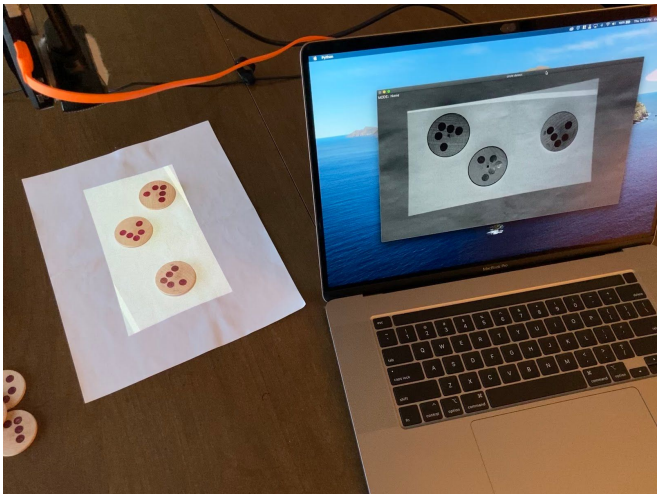
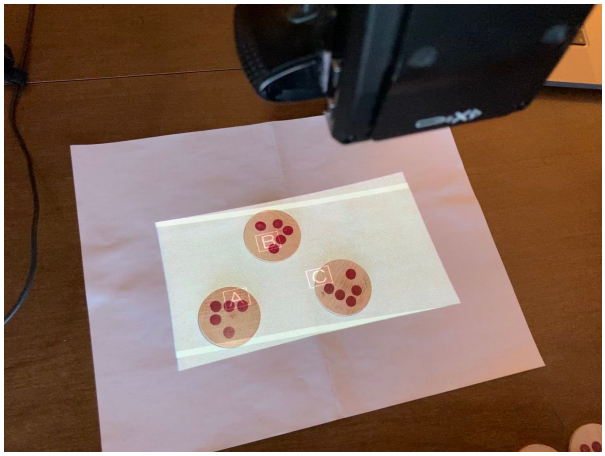


Figure 2.2.2 (on left). Shows the camera projector setup over the projected screen window
Figure 2.2.3 (on right). Shows the system hooked up to my computer with the display window on screen.

Above, in Figure 2.2.2 and Figure 2.2.3, is the layout of how the camera-projector system is set up in conjunction with my computer. The left picture shows how the USB webcam is attached to the top of the small projector and how the wooden tokens are positioned on a piece of paper. The small lit rectangle on the paper is the projected image. The picture on the right shows the control window on my computer which is the displayed image from the webcam. In this window, the user can change which mode they are in (Add Node mode, Connect Nodes mode, or

Find Path mode) and see a drawn representation of the circle that the hough circle detection found.

The program begins by initializing a video capture, the output screen windows, and the graph object that will be used. It then begins an infinite loop. Within this loop, the program converts the input image to grayscale, applies a median blur to it, and sends the image into the OpenCV function `HoughCircles()`. This function uses a method called Hough Gradient to find circles in an image. The function returns a 2-dimensional array filled with the radius value as well as the x and y coordinates for each detected circle. Within this loop of circles, the program creates a node object and stores the value as well as the x and y coordinates for all circles that were selected by the user.



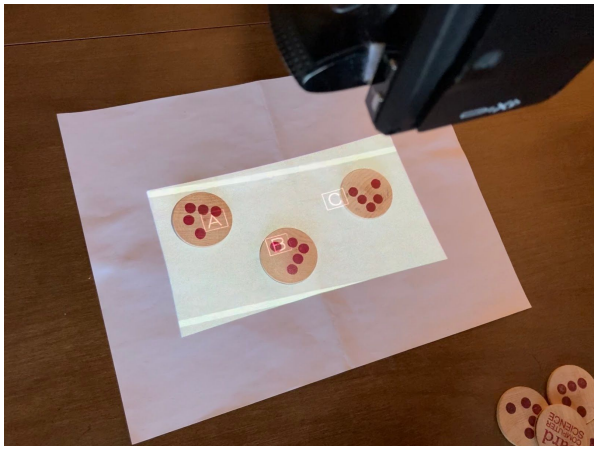


Figure 2.2.4 (on left). Shows how the node values are displayed on the projector window
Figure 2.2.5 (on right). Shows the projected nodes after the tokens have been moved around

To track the nodes between frames, I utilize the (x, y) coordinate returned by the hough circle transform. So, within this loop over all detected circles, I check if any of these newly found circles are within 1.5x the radius of any of the previously stored nodes. If so, these newly found (x, y) coordinates are stored within the near node object as its new center values. I chose

1.5x the radius because I knew that even if two tokens are right next to each other and touching, each of their centers would be around 2x the radius of one of the circles apart. I chose 1.5x the radius to account for some potential error.

The program then loops over every stored Node object in the Graph, draws it on the projected screen, and draws all connections between nodes as well.

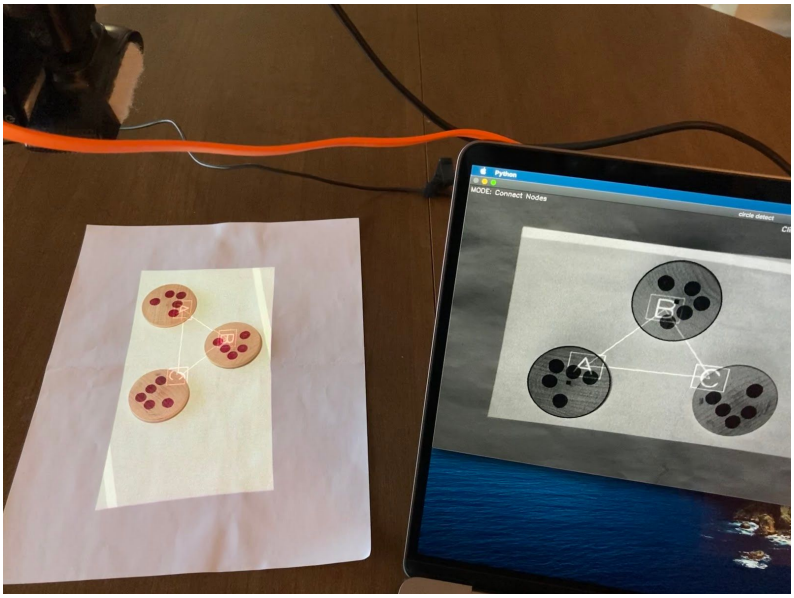


Figure 2.2.6. Shows how the projector displays the nodes once connected to each other

I use the stored x and y coordinates in the node object to draw its value on the output screen. Within the loop, I also store a list of the node's neighbors in order to keep track of connections in the Graph. The rest of the infinite loop performs specific tasks to each mode the program could be in, updates all windows, and processes key inputs as well as mouse clicks.

2.3 Just Camera Graph Program

At this point in my project, I was forced to rethink my path due to the Covid-19 pandemic. My plans from here were to use a much larger projector to make this graph detection program work on a larger surface. This would allow larger groups of people to interact with the graph data structure. However, I went back home to St. Louis and was unable to continue on this track. From here, I decided to refine the camera-projector graph program and take away the projector functionalities. I implemented a minimum spanning tree algorithm on the code I had and added some features such as deleting nodes and edge weights.

2.4 Minimum Spanning Tree Algorithm

2.4.1 Definition of Minimum Spanning Tree

Given an undirected and connected graph G, a spanning tree of G is a tree that spans G and is a subgraph of G. A tree spans a graph if and only if it connects all vertices in G. A tree is a subgraph of a graph if and only if all of its edges belong to the graph. Also, by definition, a tree does not contain any cycles. The cost of a spanning tree is the sum of all edge weights contained in the tree.

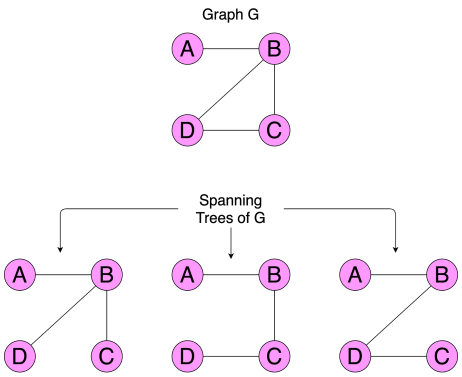


Figure 2.4.1. A graph G and all of its possible spanning trees

A *minimum spanning tree* is the tree in a weighted, undirected, and connected graph that has the smallest total edge weight cost out of all possible spanning trees.

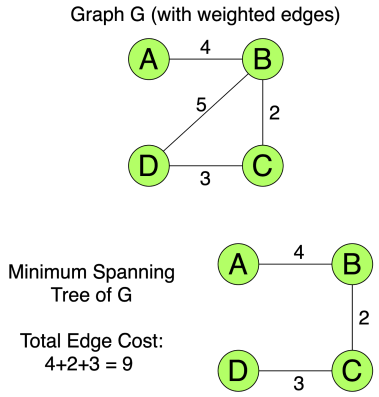


Figure 2.4.2. A weighted and undirected graph G with its minimum spanning tree

2.4.2 Kruskal's Algorithm

2.4.2.1 General Idea

Kruskal's Algorithm begins by sorting all of the edges in terms of their respective weights in ascending order. It then starts to build the MST by choosing the edge with the lowest weight cost and adding it to a result spanning tree. It proceeds to pick the edge with the next smallest weight and adds it to the result. If this next edge does not cause a cycle in the spanning tree built up so far, it stays. If it causes a cycle, then it is discarded. This process of choosing the next edge in the list of sorted edge weights and adding it to the resulting graph is repeated until the resulting spanning tree has a number of edges totaling to one less than the total number of vertices in the original graph. This will result in a minimum spanning tree of the original graph.

2.4.2.2 Example of Building an MST

Consider the following weighted and undirected graph:

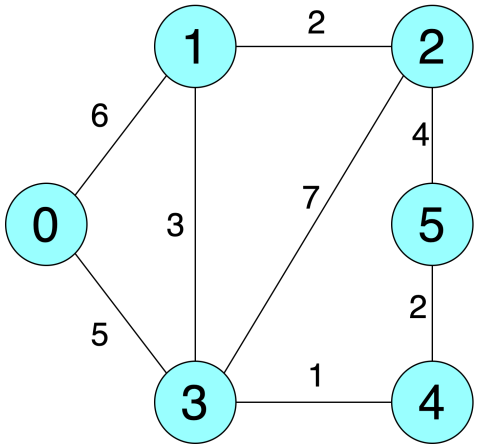
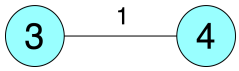


Figure 2.4.2.2. A graph for minimum spanning tree example

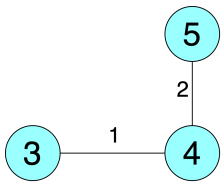
Sorted Edge List	
Edge	Weight
3 — 4	1
4 — 5	2
1 — 2	2
1 — 3	3
2 — 5	4
0 — 3	5
0 — 1	6
2 — 3	7

Figure 2.4.2.3. Sorted edge list for graph

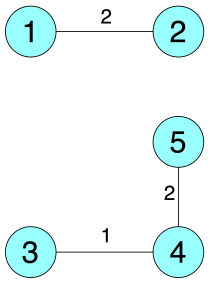
1. Pick edge 3-4 from the sorted edge list since it has the smallest edge cost. It does not form a cycle since it is the first edge added to the spanning tree, so keep it.



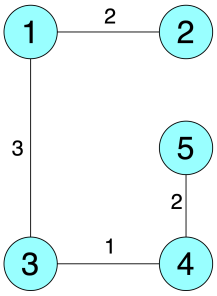
2. Next, pick edge 4-5. No cycle is formed so include it in the final spanning tree.



3. Pick edge 1-2. No cycle is formed so add it to the final spanning tree.



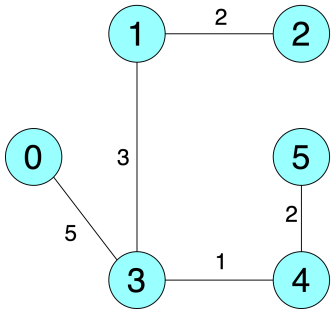
4. Pick edge 1-3. No cycle formed. Add it to the final spanning tree.



Sorted Edge List	
Edge	Weight
3 — 4	1
4 — 5	2
1 — 2	2
1 — 3	3
2 — 5	4
0 — 3	5
0 — 1	6
2 — 3	7

5. Pick edge 2-5. This edge creates a cycle in the spanning tree. Therefore, discard it.

6. Pick edge 0-3. No cycle formed. Add it.



At this point, since the total number of edges in this resulting spanning tree is equal to one less than the number of vertices in the original graph, we know this is the minimum spanning tree.

2.4.3 How I Implemented Kruskal's Algorithm

In my *graph_detect.py* program (Refer to Appendix), I used a graph object to store an array of multiple node objects. The Graph class has two member functions; `add()` and `delete()`. The `add()` function is used to append a node to the list of nodes as well as adding one to the total number of vertices in the graph. The `delete()` function deletes a specified node from the Graph. The node class stores the value of the node, the x and y coordinates of the node in the image, and a list of neighbor nodes. With this method of storing the graph object and various node objects, I needed to create a function that builds a list of edges and their corresponding weights from this representation. For this function, I loop through every node object in the graph. For each node, I loop through its neighbors. I get the euclidean distance between the initial node and its neighbor. If the edge does not exist in the array of edges already, I add the node value, neighbor node value, and weight as one array to a larger array of all the edges in the graph. For example, if I have one node with value A, a neighbor with value B, and a weight value of 10 between them, I will add the array `['A', 'B', 10]` to an array of `edge_weights`. The final edge weights array will look something like `edge_weights = [['A', 'B', 10], ['B','C', 15], ...]`.

I used *geeksforgeeks.org* [7] as a reference for this algorithm. With this built up array of edge weights, I then can begin to implement Kruskal's algorithm. The first step is to sort this array of edges. After sorting, I initialize three different arrays that keep track of parent nodes and ranks. This is useful when figuring out if adding an edge to the spanning tree creates a cycle. Then, I loop from 0 to (V-1) where V is the number of vertices in the graph. This is because once we add V-1 edges to the spanning tree, we know we have completed the minimum spanning tree.

Within this loop, we extract the smallest edge from the list of edge weights. From here, I find if the nodes share a root parent node. To do this, I use a function called `find()` that takes in an array of all the initial node values, another array of corresponding parent values for each node, and the value of the node whose root parent we are trying to find. This finds the root parent by parsing through the two arrays with recursion. Once these parents are found, if both node's root parents are equivalent, then we know the edge we are processing creates a cycle. If this is the case we move on to the next smallest edge in the list of edge weights. If the edge does not create a cycle, I append the edge to the result array and then update the arrays that are holding the root parent values by calling a function named `union()` which finds the root of both nodes parent values and performs union by rank (attaches smaller rank tree under the root of the higher rank tree). This ensures that the parent array that is used for finding cycles is up to date and can identify cycles when processing the next edge. This process is repeated while the loop is active. The result array will contain all of the edges in the minimum spanning tree and their corresponding weights.

2.4.4 Applications of MST [8]

There are many applications of the Minimum Spanning Tree Problem. MST can be useful in designing various networks such as phone, internet, electrical, roads, etc. In all of these cases, there are both financial and spatial benefits to having a connection between all nodes in a graph/network with the shortest total distance. Utilizing an MST (minimum spanning tree) solution to these problems would also eliminate redundancies in a network by ensuring there are no cycles in the connections.

Another application of Minimum Spanning Tree is the Travelling Salesperson problem. This problem is NP-hard, meaning it has no known polynomial-time solution. The main idea is that there is a set of cities and known distances between each pair of cities. The problem seeks to find the shortest route that visits every city exactly once and returns to the initial city in the route. MST can be useful in approximating a solution for the travelling salesperson problem. We can apply the triangle inequality (the sum of the lengths of two sides of a triangle is always greater than the length of the third side) to a graph representation of the TSP problem. This is especially useful when finding a solution to the problem using an MST algorithm. According to geeksforgeeks.org, "The least distant path to reach a vertex j from i is always to reach j directly from i, rather than through some other vertex k (or vertices), i.e., dis(i, j) is always less than or equal to dis(i, k) + dist(k, j)" [9]. This rule ensures that the output of an MST algorithm will never be worse than twice the cost of an optimal tour of the TSP problem.

Minimum spanning trees can also be useful for cluster analysis. One can find a solution to a k clustering problem by constructing an MST with the nodes, then eliminating the k-1 largest edges in the resulting MST. This creates k clusters with maximized distance between each different cluster [10].

2.5 OpenCV Text Detection

At the beginning of this project, I was considering creating a programming workspace for a user to physically interact with the code they were writing. This could be done by using OCR technologies to interpret handwritten code. I utilized a tutorial on *pyimagsearch.com* written by Adrian Rosebrock that covers text detection using OpenCV and Python [5]. The article included

code specifically designed to not only detect the usual black text on a white background, but also natural scene text like billboards or other lettered words in public. This can be a difficult task because an image with natural scene text has the potential to contain noise, heavy blurring, odd light conditions, unparallel viewing angles, etc. The code in OpenCV utilizes a deep learning text detector known as EAST (An Efficient and Accurate Scene Text Detector). This deep learning method was developed and published in a paper from 2017 written by Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang [6]. After implementing this code and testing how it worked with my webcam, I realized text detection can be very unreliable. I proceeded to do more research to try and find a better solution to this.

2.6 Google Cloud Vision API OCR

After continuing my research on other methods of text detection, I found Google's Cloud Vision API client libraries. They offer an OCR (Optical Character Recognition) tool that can identify text within an image. With this option, I was able to use Google's Cloud computing to process my images that contained text. To do this, I first had to initialize an `ImageAnnotatorClient()` object to make the API request. I then used the `document_text_detection()` function that is available as a method of the client object. This takes in an image then returns a hierarchical representation of the text detected in the image.

Below are two examples of the output of this program:

```
Block confidence: 0.9100000262260437

    Paragraph confidence: 0.949999988079071
        Word text: def (confidence: 0.9900000095367432)
        Word text: square (confidence: 0.9900000095367432)
        Word text: ( (confidence: 1.0)
        Word text: x (confidence: 0.5600000023841858)
        Word text: ) (confidence: 0.9599999785423279)
        Word text: : (confidence: 0.9399999976158142)
    Paragraph confidence: 0.8600000143051147
        Word text: return (confidence: 0.9900000095367432)
        Word text: X (confidence: 0.4399999976158142)
        Word text: * (confidence: 0.9300000071525574)
        Word text: X (confidence: 0.41999998688697815)
----------------------------------------

final output text:

def square (x):
return X*X

----------------------------------------
```

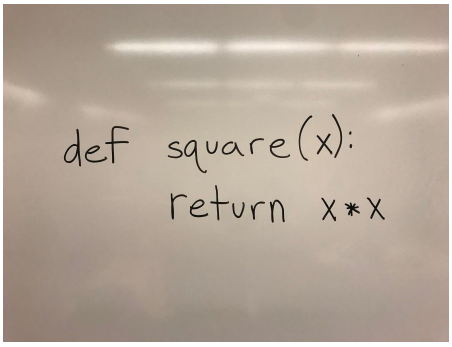


Figure 2.3.1 (on left). The output text displayed in the terminal.
Figure 2.3.2 (on right). The hand-written code chunk that is fed as input into the program.

```
Block confidence: 0.9300000071525574

    Paragraph confidence: 0.9800000190734863
        Word text: def (confidence: 0.9900000095367432)
        Word text: sum (confidence: 0.9900000095367432)
        Word text: ( (confidence: 0.9800000190734863)
        Word text: input (confidence: 0.9900000095367432)
        Word text: ) (confidence: 0.9900000095367432)
        Word text: : (confidence: 0.8500000238418579)
    Paragraph confidence: 0.8999999761581421
        Word text: total (confidence: 0.9900000095367432)
        Word text: = (confidence: 0.8600000143051147)
        Word text: 0 (confidence: 0.3700000047683716)
        Word text: for (confidence: 0.9900000095367432)
        Word text: x (confidence: 0.5400000214576721)
        Word text: in (confidence: 0.9200000166893005)
        Word text: input (confidence: 0.9300000071525574)
        Word text: : (confidence: 0.9399999976158142)
    Paragraph confidence: 0.9200000166893005
        Word text: total (confidence: 0.9900000095367432)
        Word text: + (confidence: 0.44999998807907104)
        Word text: = (confidence: 0.9100000262260437)
        Word text: x (confidence: 0.25999999046325684)
        Word text: return (confidence: 0.9900000095367432)
        Word text: total (confidence: 0.9900000095367432)
----------------------------------------

final output text:

def sum (input):
total=0 for x in input:
total += x      return total

----------------------------------------
```

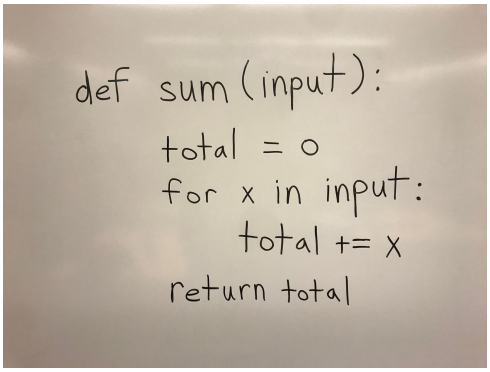


Figure 2.3.3 (on left). The output text displayed in the terminal.
Figure 2.3.4 (on right). The hand-written code chunk that is fed as input into the program.

The images on the right in both of the above examples (Figure 2.3.2 and Figure 2.3.4) contain an example of the hand written functions in python that are input into the programs. The text to the left of each corresponding block of code is the output printed to the terminal window from the API request (Figure 2.3.1 and Figure 2.3.3). The Google Cloud API OCR tended to be much more accurate than the EAST text detection I implemented in OpenCV. The Google OCR method also captured new lines, spaces, and other types of whitespace. It also keeps track of each symbol's x and y coordinates in the image. With the output text from the OCR, I then build up a string containing all of the symbols that the OCR found and feed that into an output python file. This had some downfalls as the tabs from the hand written code were not found very often in my testing. Since python depends upon whitespace, this was an issue. However, this could be overcome by utilizing the coordinates of each word to detect if there is a tab present using a threshold of sorts. Although this initial endeavour into creating an interactive programming workspace did not pan out, I would still like to potentially delve into this area in the future.

3. Results

3.1 Camera Graph Detection Results

Figure 3.1.1. Initial setup for graph detection program

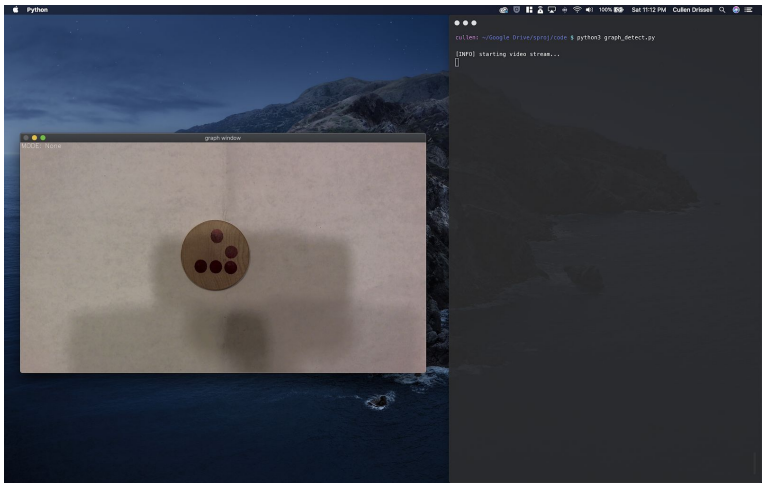


Figure 3.1.2. Initial screen display for graph detection program

Figure 3.1.1 and Figure 3.1.2 show what the initial setup would look like when running the graph detection program. The USB webcam is hooked up to the computer and is transferring in the video feed.

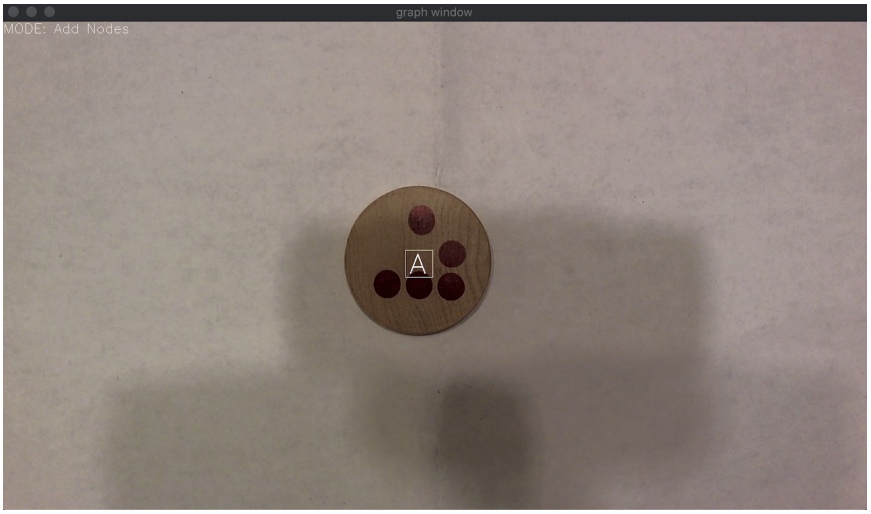


Figure 3.1.3. Graph window displaying one added node with value ‘A’

In Figure 3.1.3, we can see what it would look like when the user adds a node with value ‘A’. To add a node while running the program, the user would hit the ‘a’ key. Once the displayed mode, which is found in the top left of the graph window, says that it is ‘Mode: Add Nodes’, then the user can click on a displayed node that they want to assign a value to. This will create a node object and assign it the entered value.

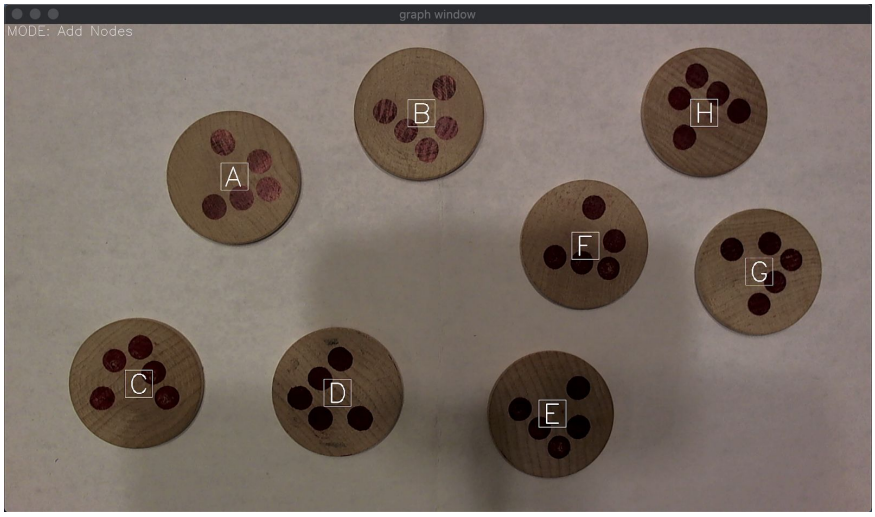


Figure 3.1.4. Graph window displaying multiple added nodes

Figure 3.1.4 shows what the screen may look like if the user adds eight nodes to the graph.

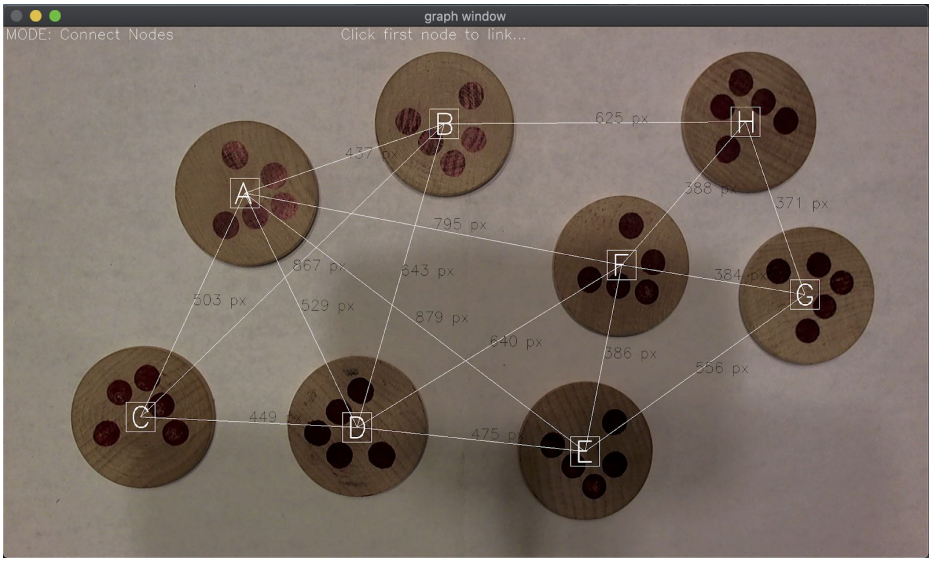


Figure 3.1.5.Graph window displaying all of the added nodes and their connections

After a number of nodes are added, the user can then change the mode to ‘Connect Nodes’ by hitting the ‘c’ key. This mode allows the user to click on two nodes to add a connection between them. The distance (number of pixels) between the two nodes is also recorded and displayed. This value is used for the Minimum Spanning Tree algorithm.

The user has, at any time, the option to delete a node as well by switching into ‘Delete Node’ mode by hitting the ‘d’ key. Figure 3.1.6 shows the previous image with the ‘F’ node deleted. By deleting a node, the Node object in the graph is deleted as well as all of its connections to other nodes.

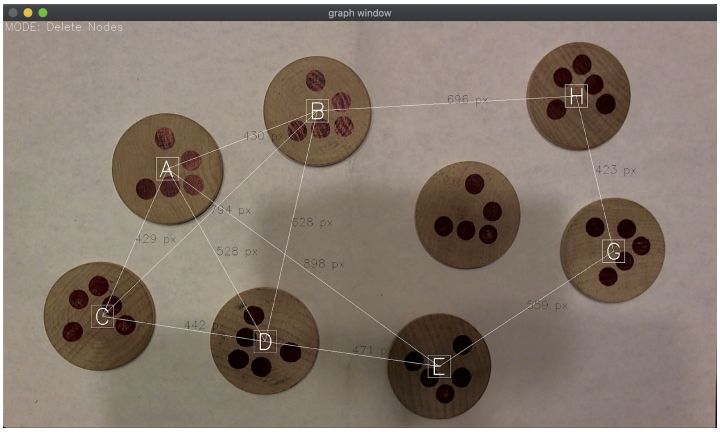


Figure 3.1.6. Graph window display after node with value ‘F’ is deleted

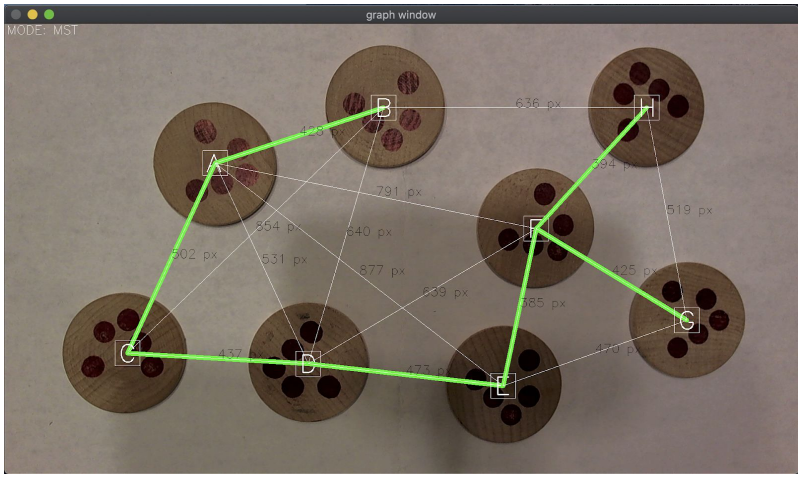


Figure 3.1.7. Graph window displaying the found MST

Once all of the nodes the user wants in the graph are connected, they can now hit the ‘m’ key to find the minimum spanning tree in the current graph. The green connections show the minimum spanning tree in the graph.

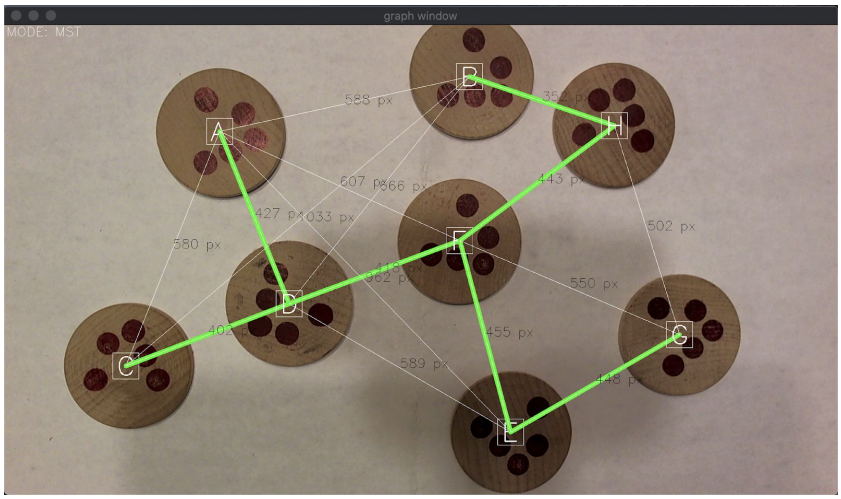


Figure 3.1.1.Graph window displaying the MST after some nodes are move around

Figure 3.1.1 shows the same exact graph but with a few nodes shifted around. As you can see, the minimum spanning tree in the graph adapted to the node location changes. Since the edge weights in the graph are defined as the euclidean distance between nodes, changing the shape of the graph changes the MST in the graph.

3.2 Camera Graph Detection Limitations

There were a few issues I ran into with this circle detection graph program. Firstly, there were two cases that would break important features in the program while running; the circle detection breaking and the circle tracking breaking.

The circle detection relies fully on the webcam, so if the image was not clear or had noise, the detection would be finicky. One of the main issues I had was with lighting. This was especially important when using the projector. The projected light on the graph surface affected the webcam's image. The ambient light in the room also influenced this image. To combat this, I needed to change the shade of gray that was displayed behind the graph node tokens on the surface. If the ambient light in the room was darker, there needed to be a darker gray background behind the tokens and if the room's ambient light were brighter, the background on the surface needed to be lighter. The focus of the camera also affected this detection as well. If the image was blurry, the detection was not as accurate. I also was unable to fix the focus of the camera because I needed the distance between the camera and the surface to be dynamic. This meant that the webcam would go blurry at random times. I would use my hand under the camera to bring it back into focus on the token surface if this happened.

The other issue that came up sometimes in the program was the circle tracking breaking. Since the circles were being tracked frame by frame by checking newly detected circles positions and comparing this new position to all stored positions, if tokens were moved too fast between frames, this tracking mechanism would miss the token that was previously assigned to the Node in question. This would cause the Node object to not be assigned to any token in particular. If

this happens, the user can move the empty token back to where it was previously located and the Node object will be reassigned to the token.

4. Conclusion

4.1 Summary

This project took many twists and turns over the past year. It started out as a workspace for a programmer to more easily visualize their code and work with their hands. It then transitioned into a more educational experience where a user could hand-write code and data structures. The final idea consisted of using a camera-projector system to create an interface that allows one or multiple users to interact physically and dynamically with data structures. Due to the COVID-19 pandemic, this project was not able to flourish to its full potential. After leaving New York and traveling home, I was unable to use all of the resources that were available to me at Bard. Although I am proud of the final product that came from this project, I had goals that were unable to be achieved.

4.2 Motivation

I have always been interested in teaching and education. I spent about three years instructing trumpet to younger students in my school district during high school. I have some experience tutoring math for middle schoolers as well. I co-founded a four week summer camp for elementary aged kids that focussed on social justice and community activism in my hometown. More recently in 2018, I worked as a computer science tutor for Bard. I enjoyed

these experiences so much and learned a lot about productive and efficient methods of teaching. When I came to Bard, I didn't know what I wanted to do at all. My main interests throughout high school were physics and music. Choosing to major in computer science was a last minute decision upon coming to Bard. One interest that has stayed consistent throughout all this time is education. I have taken two classes on education and teaching at Bard; Getting Schooled in America and Identity in the Classroom. Considering all of this, it is fitting that my senior project revolves around topics of education and learning. My project's goal is to make the process of learning to code easier and more accessible. For some individuals, it can be hard to create a mental image of how data structures and algorithms interact with each other. By creating a camera system that helps make learning the graph data structure easier, dynamic, and more interactive, I hope to make the general process of learning how to program smoother for individuals.

4.3 Progression of the Project

This project initially sought to create a dynamic and educational way to interact with programs, data structures, and algorithms. When I began working on this project, my initial idea was to create an interactive, visual workspace for a user to more easily learn programming. My plan was to create connections between an IDE, a camera, as well as pen and paper. I wanted the user to be able to write code by hand and through OCR technologies, have their hand-written code be processed and input into a program they're writing. I also wanted the user to have the ability to draw data structures from scratch and have the camera process these into a program as well.

Over time, the project changed and adapted. I continued with the idea of keeping the final product educational, yet, I wanted to add an aspect of collaboration and group work. I then decided to create an environment where a user could construct data structures using physical tokens on a surface then have a camera system detect the data structure and display values upon them using a projector. This would allow the user, or multiple users, to physically manipulate a data structure and watch how various algorithms interact with it in real time. I eventually decided to implement a minimum spanning tree algorithm in this program. This tool would be helpful in educational settings especially when teaching about data structures and algorithms.

4.4 Future Work

If I were to continue this work, there are many things that I would consider implementing. When I began this project, my initial idea was to create a workspace for a user working on a program. I wanted to tie together an IDE (Integrated Development Environment), an image window that processes hand written code, and a window where the user could manipulate data structures by hand. Below, in Figure 4.4.1, is a mockup of what I had in mind for the final product.

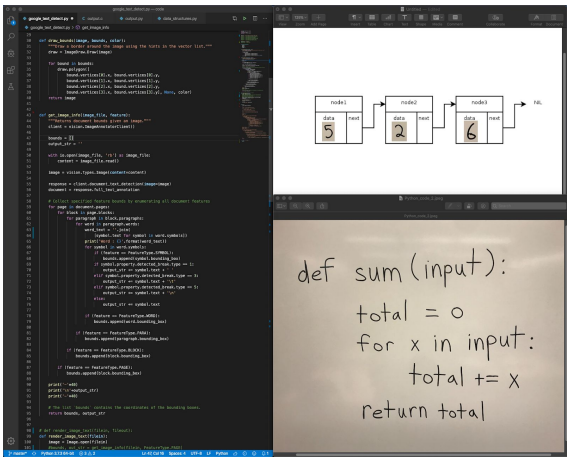


Figure 4.4.1. A mock-up for my initial idea for this project

In Figure 4.4.1, the IDE is on the left with an example of what the code might look like. In the top right, there is a displayed linked list data structure with hand-written, user defined values for each node. In the bottom right, there is a video feed from a webcam that detects and processes a hand written chunk of code. If I were to continue working in this realm of creating educational tools for coding, I would surely delve deeper into this idea of a programming workspace.

Other features I would like to include in future work on this project would be to use a larger projector to cover more area for the workspace in the projector-camera system. With my current setup, I could only fit about 6-7 nodes within the projected region. With a larger area to work with, many users could interact with the graph at once creating a collaborative educational experience. This larger area would also call for more refined camera vision techniques as well. If I were to use a larger projector and workspace, I would want to include a homography between the windows of the camera and the projector to better line up the projected images on the display window. This would also allow for the camera to be set up at varying angles relative to the projected image. Another feature I would implement if I were to carry on this project would be to program better camera calibration for the OpenCV hough circle transform. There are many other arguments that can be fed into the `HoughCircles()` function that OpenCV offers. There is a potential to change the upper and lower threshold for the Canny edge detector that is applied to the input image. There is an option to manipulate the threshold for the center detection. There is also the ability to change the minimum and maximum radius of the potential detected images which could optimize performance by lowering the amount of storage necessary for keeping track of detected circles in the image. If I were to continue this work, I would figure out a way to

optimize these values and potentially make them dynamic as the program is running. Another feature I would like to add is a better and more optimized front end. OpenCV offers some GUI features but I would like to make a visually appealing GUI for this project in the future. Overall, the final product of this project, a functional graph detection camera system that allows a user to manipulate a graph data structure in real time, acts as an educational tool that can potentially help users better visualize and learn how algorithms interact with data structures.

Bibliography

[1] Dynamicland, dynamicland.org, 2018.

[2] Kosminsky, Eli. "Dynamicland: A New Direction for Immersive Simulations." *The Concord Consortium*, 23 Mar. 2018, concord.org/blog/dynamicland-a-new-direction-for-immersive-simulations/.

[3] Guo, Philip. "Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education." *ACM Technical Symposium on Computer Science Education (SIGCSE),* 2013.

[4] Bradski, G. "The OpenCV Library." *Dr. Dobb's Journal of Software Tools,* 2000.

[5] Rosebrock, Adrian. "OpenCV Text Detection (EAST text detector)." *pyimagesearch,* 20 Aug. 2018. https://www.pyimagesearch.com/2018/08/20/opencv-text-detection-east-text-detector/.

[6] Zhou et al. "EAST: An Efficient and Accurate Scene Text Detector." *CoRR,* 2017. https://arxiv.org/abs/1704.03155.

[7] Barnwal, Aashish. "Kruskal's Minimum Spanning Tree Algorithm | Greedy Algo-2." *GeeksforGeeks,* 2019. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

[8] "Applications of Minimum Spanning Tree Problem." *GeeksforGeeks.*

[9] "Travelling Salesman Problem: Set 2 (Approximate Using MST)." *GeeksforGeeks*, 2018, www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/?ref=rp.

[10] Kingsford, Carl. "Minimum Spanning Trees & Clustering." *Department of Computer Science University of Maryland, College Park*

Appendix

Program 1: graph_detect.py

This program is the final product of the camera system detecting circles for graph manipulation. It is very similar to the camera-projector system program. The only difference is that it displays the node values and connections on the computer screen rather than the projected screen.

```
import numpy as np
import cv2
import imutils
from imutils.video import FPS

DEBUG = False
webcam_x_dim = 2592
webcam_y_dim = 1944
ADD_NODE_FLAG = False
CONNECT_NODES_FLAG = False
NODE_CLICKED_COUNT = 0
FIND_PATH_FLAG = False
FIND_MST_FLAG = False
DELETE_NODE_FLAG = False
click_x = 0
click_y = 0

#-------------------------------------------------------------
# Classes

class Node():

   def __init__(self, input_val, x_pos, y_pos, neighbors=None):
       self.value = input_val
       self.x = x_pos
       self.y = y_pos
       if neighbors is None:
           self.neighbors = []
       else:
           self.neighbors = neighbors

   def add_neighbor(self, neighbor):
       self.neighbors.append(neighbor)

   def update_pos(self, new_x, new_y):
```

```
        self.x = new_x
        self.y = new_y


class Graph():

    def __init__(self,g=[]):
        self.list = g
        self.V = 0

    def add(self, node):
        self.list.append(node)
        self.V += 1

    def delete(self, node):
        for neighbor in node.neighbors:
            if node in neighbor.neighbors:
                neighbor.neighbors.remove(node)
        self.list.remove(node)
        self.V -= 1

#---------------------------------------------------------------
# Functions for OpenCV

'''
Function to handle mouse clicks on opencv screen
'''
def on_click(event, x, y, flags, param):
    global click_x, click_y, ADD_NODE_FLAG, FIND_PATH_FLAG, NODE_CLICKED_COUNT
    if event == cv2.EVENT_LBUTTONDOWN:
        click_x = x
        click_y = y
        if CONNECT_NODES_FLAG:
            if DEBUG:
                print('Clicked x: '+str(x)+', y: '+str(y)+'\tTo connect nodes')
            NODE_CLICKED_COUNT += 1
        elif FIND_PATH_FLAG:
            if DEBUG:
                print('Clicked x: '+str(x)+', y: '+str(y)+'\tTo find node path')
            NODE_CLICKED_COUNT += 1
        elif FIND_MST_FLAG:
            if DEBUG:
                print('Clicked x: '+str(x)+', y: '+str(y)+'\tTo find MST')
        elif ADD_NODE_FLAG:
            if DEBUG:
                print('Clicked x: '+str(x)+', y: '+str(y)+'\tTo add node')
        elif DELETE_NODE_FLAG:
            if DEBUG:
                print('Clicked x: '+str(x)+', y: '+str(y)+'\tTo delete node')
```

```
def drawLinkDist(screen, node1, node2):
    distance = dist(node1.x, node1.y, node2.x, node2.y)
    if node1.x <= node2.x:
        text_pos_x = node1.x+((node2.x-node1.x)/2)
    else:
        text_pos_x = node1.x-((node1.x-node2.x)/2)
    if node1.y >= node2.y:
        text_pos_y = node1.y-((node1.y-node2.y)/2)
    else:
        text_pos_y = node1.y+((node2.y-node1.y)/2)
    cv2.putText(screen,
                str(int(distance))+' px',
                (int(text_pos_x), int(text_pos_y)),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0))


#-------------------------------------------------------------

# Functions for graph algorithms

"""
Function for building a 2d array of edges in graph
Form:
    [[node1, node2, weight(node1 -> node2)], etc.]
"""
def build_edges(graph):
    edge_weights = []
    for node in graph.list:
        for neighbor in node.neighbors:
            weight = dist(node.x, node.y, neighbor.x, neighbor.y)
            if [neighbor.value, node.value, weight] not in edge_weights:
                edge_weights.append([node.value, neighbor.value, weight])
    return edge_weights

"""
Function for recursively finding the root parent of a node val
"""
def find(orig_parent_array, parent, val):
    idx = orig_parent_array.index(val)
    if parent[idx] == val:
        return val
    return find(orig_parent_array, parent, parent[idx])

"""
A function that does union of two sets of x and y
(uses union by rank)
"""
def union(orig_parent_array, parent, rank, x, y):
    xroot = find(orig_parent_array, parent, x)
```

```
    yroot = find(orig_parent_array, parent, y)

    xroot_idx = parent.index(xroot)
    yroot_idx = parent.index(yroot)

    # Attach smaller rank tree under root of
    # high rank tree (Union by Rank)
    if rank[xroot_idx] < rank[yroot_idx]:
        parent[xroot_idx] = yroot
    elif rank[xroot_idx] > rank[yroot_idx]:
        parent[yroot_idx] = xroot

    # If ranks are same, then make one as root
    # and increment its rank by one
    else:
        parent[yroot_idx] = xroot
        rank[xroot_idx] += 1

    return parent, rank


def kruskalMST(graph):

    result = []  # This will store the resultant MST

    i = 0  # An index variable, used for sorted edges
    e = 0  # An index variable, used for result[]

    edges = build_edges(graph)

    # Sort all the edges in non-decreasing
    # order of their
    # weight.  If we are not allowed to change the
    # given graph, we can create a copy of graph
    edges = sorted(edges, key=lambda item: item[2])

    orig_parent_array = []
    parent = []
    rank = []

    # Create V subsets with single elements
    for node in graph.list:
        orig_parent_array.append(node.value)
        parent.append(node.value)
        rank.append(0)

    # Number of edges to be taken is equal to V-1
    while e < graph.V - 1:

        # Pick the smallest edge and increment
```

```
        # the index for next iteration
        u, v, w = edges[i]
        i = i + 1
        x = find(orig_parent_array, parent, u)
        y = find(orig_parent_array, parent, v)

        # If including this edge doesn't cause cycle,
        # include it in result and increment the index
        # of result for next edge
        if x != y:
            e = e + 1
            result.append([u, v, w])
            parent, rank = union(orig_parent_array, parent, rank, x, y)
        # Else discard the edge

    if DEBUG:
        # print the contents of result[] to display the built MST
        print('Following are the edges in the constructed MST')
        for u, v, weight in result:
            print(str(u)+' -- '+str(v)+' == '+str(weight))

    return result


"""
Returns euclidean distance between point one (x1,y1) and point two (x2,y2)
"""
def dist(x1, y1, x2, y2):
    return np.sqrt(((y2-y1)**2)+((x2-x1)**2))


"""
Simple DFS algorithm for finding first path between 2 nodes
"""
def find_graph_path(graph, start_node, end_node, path=[]):
    path = path + [start_node.value]
    if start_node.value == end_node.value:
        return path
    for neighbor in start_node.neighbors:
        if neighbor.value not in path:
            new_path = find_graph_path(graph, neighbor, end_node, path)
            if new_path:
                return new_path
    return None

#-----------------------------------------------------------------

def main():

    global DEBUG, ADD_NODE_FLAG, CONNECT_NODES_FLAG, FIND_PATH_FLAG, \
```

```
FIND_MST_FLAG, NODE_CLICKED_COUNT, DELETE_NODE_FLAG, click_x, click_y
DRAW_PATH_FLAG = False
initial_connect_node = None

# begins video stream
print("[INFO] starting video stream...")
cap = cv2.VideoCapture(0)

if not cap.isOpened():
    print('Error initializing camera. Exiting...')
    return

# begins FPS
fps = FPS().start()

# initialize graph
graph = Graph()

path = []

cv2.namedWindow('graph window')
cv2.setMouseCallback('graph window', on_click)


# circle detection loop
while cap.isOpened():

    # reads in a frame from video stream
    ret, frame = cap.read()

    # if no video capture, breaks out of while loop
    if not ret:
        break

    output_screen = frame.copy()

    # gets key press
    key = cv2.waitKey(1) & 0xFF

    # converts frame to gray for hough circle detection
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # gets key press
    key = cv2.waitKey(1) & 0xFF

    # blurs image to avoid noise
    gray = cv2.medianBlur(gray, 5)

    # gets key press
    key = cv2.waitKey(1) & 0xFF
```

```
circles = cv2.HoughCircles(gray, cv2.HOUGH_GRADIENT, 1.2, 100)

# gets key press
key = cv2.waitKey(1) & 0xFF

if circles is not None:
    # convert the (x, y) coordinates and radius of the circles to integers
    circles = np.round(circles[0, :]).astype("int")

    for (circle_x, circle_y, circle_r) in circles:

        #print('circle radius: '+str(circle_r))
        if DEBUG:
            cv2.circle(gray, (circle_x, circle_y), circle_r, (0, 255, 0), 2)
            cv2.rectangle(gray, (circle_x - 5, circle_y - 5),
                          (circle_x + 5, circle_y + 5), (0, 128, 255), -11)


        # Adds a new node to graph
        if ADD_NODE_FLAG:
            if DEBUG:
                print('to add: circle x and y: '+str(circle_x) +' '+ str(circle_y))
                print('to add: click x and y: '+str(click_x) +' '+ str(click_y))
            if (circle_x-circle_r) < click_x < (circle_x+circle_r) and \
                (circle_y-circle_r) < click_y < (circle_y+circle_r):
                # checks if another node already has this x, y position
                count = 0
                for node in graph.list:
                    if (circle_x-circle_r) < node.x < (circle_x+circle_r) and \
                        (circle_y-circle_r) < node.y < (circle_y+circle_r):
                            count += 1
                if count == 0:
                    node_val = input('\nEnter a value for added node: ')
                    graph.add(Node(node_val, circle_x, circle_y))
                    #ADD_NODE_FLAG = False

        elif DELETE_NODE_FLAG:
            if DEBUG:
                print('to delete: circle x and y: '+str(circle_x) + \
                      ' '+ str(circle_y))
                print('to delete: click x and y: '+str(click_x) +' '+ str(click_y))
            if (circle_x-circle_r) < click_x < (circle_x+circle_r) and \
                (circle_y-circle_r) < click_y < (circle_y+circle_r):
                # checks if a node exists where user clicked. if so, deletes node
                for node in graph.list:
                    if (circle_x-circle_r) < node.x < (circle_x+circle_r) and \
                        (circle_y-circle_r) < node.y < (circle_y+circle_r):
                            print('\ndeleting node '+str(node.value)+'...')
                            graph.delete(node)
```

```
        for node in graph.list:
            # checks if node has moved since last iteration
            if (circle_x-(1.5*circle_r)) < node.x < (circle_x+(1.5*circle_r)) and \
                (circle_y-(1.5*circle_r)) < node.y < (circle_y+(1.5*circle_r)):
                node.update_pos(circle_x, circle_y)

# draws the node values and connections between nodes
for node in graph.list:
    cv2.rectangle(output_screen,
                (node.x - 30, node.y - 30),
                (node.x + 30, node.y + 30),
                (255, 255, 255), 1)
    cv2.putText(output_screen, node.value, (node.x-21, node.y+21),
                cv2.FONT_HERSHEY_SIMPLEX, 2, (255, 255, 255), 2)
    for linked_node in node.neighbors:
        if FIND_MST_FLAG:
            mst = kruskalMST(graph)
            for edge in mst:
                if edge[0] == node.value and edge[1] == linked_node.value:
                    cv2.line(output_screen,
                        (node.x, node.y),
                        (linked_node.x, linked_node.y),
                        (0, 250, 0), 10)
                    drawLinkDist(output_screen, node, linked_node)
        elif path and node.value in path and linked_node.value in path:
            if DEBUG:
                print('\n')
                print('node value: '+str(node.value))
                print('linked node value: '+str(linked_node.value))
                print(path)
                print('\n')
            if path.index(node.value) == path.index(linked_node.value) - 1:
                cv2.line(output_screen,
                        (node.x, node.y),
                        (linked_node.x, linked_node.y),
                        (200, 200, 0), 10)
                drawLinkDist(output_screen, node, linked_node)
        # draws rest of lines
        cv2.line(output_screen,
                (node.x, node.y),
                (linked_node.x, linked_node.y),
                (255, 255, 255), 1)
        drawLinkDist(output_screen, node, linked_node)

# resets node clicked count if the counter goes over
# two meaning user clicked on space where node was not
if NODE_CLICKED_COUNT > 2:
    NODE_CLICKED_COUNT = 0
```

```
if CONNECT_NODES_FLAG:
    cv2.putText(output_screen, 'MODE: Connect Nodes', (5, 25),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
    if NODE_CLICKED_COUNT == 0:
        cv2.putText(output_screen, 'Click first node to link...', (700, 25),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
    elif NODE_CLICKED_COUNT == 1:
        count = 0
        for node in graph.list:
            if (node.x-40) < click_x < (node.x+40) and \
                (node.y-40) < click_y < (node.y+40):
                count += 1
                initial_connect_node = node
                cv2.putText(output_screen, 'Click second node to link...', (700, 25),
                            cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
        if count == 0:
            NODE_CLICKED_COUNT -= 1
    elif NODE_CLICKED_COUNT == 2:
        count = 0
        for node in graph.list:
            if (node.x-40) < click_x < (node.x+40) and \
                (node.y-40) < click_y < (node.y+40):
                if node.value == initial_connect_node.value:
                    print('\nCannot link a node to itself')
                elif node.value in initial_connect_node.neighbors:
                    print('\nNodes '+str(initial_connect_node.value)+ \
                        ' and '+str(node.value)+' are already connected')
                    NODE_CLICKED_COUNT = 0
                else:
                    count += 1
                    print('\nlinked nodes '+str(node.value)+ \
                        ' and '+str(initial_connect_node.value))
                    node.add_neighbor(initial_connect_node)
                    initial_connect_node.add_neighbor(node)
                    NODE_CLICKED_COUNT = 0
        if count == 0:
            NODE_CLICKED_COUNT -= 1
elif FIND_PATH_FLAG:
    cv2.putText(output_screen, 'MODE: Find Path', (5, 25),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
    if NODE_CLICKED_COUNT == 0:
        cv2.putText(output_screen, 'Click start node in desired path...', (700, 25),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
    elif NODE_CLICKED_COUNT == 1:
        count = 0
        for node in graph.list:
            if (node.x-40) < click_x < (node.x+40) and \
                (node.y-40) < click_y < (node.y+40):
                count += 1
                start_node = node
```

```
                cv2.putText(output_screen, 'Click end node in desired path...',
                            (700, 25),
                            cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
        if count == 0:
            NODE_CLICKED_COUNT -= 1
    elif NODE_CLICKED_COUNT == 2:
        count = 0
        for node in graph.list:
            if (node.x-40) < click_x < (node.x+40) and \
                (node.y-40) < click_y < (node.y+40):
                if node.value == start_node.value:
                    print('\nCannot link a node to itself')
                else:
                    count += 1
                    if DEBUG:
                        print('finding path between node '+str(start_node.value)+ \
                            ' and node '+str(node.value))
                    path = find_graph_path(graph, start_node, node)
                    print('\npath between node '+str(start_node.value)+ \
                           ' and node '+str(node.value)+': ',path)
                    DRAW_PATH_FLAG = True
                    NODE_CLICKED_COUNT = 0
        if count == 0:
            NODE_CLICKED_COUNT -= 1
elif ADD_NODE_FLAG:
    cv2.putText(output_screen, 'MODE: Add Nodes', (5, 25),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
elif DELETE_NODE_FLAG:
    cv2.putText(output_screen, 'MODE: Delete Nodes', (5, 25),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
elif FIND_MST_FLAG:
    cv2.putText(output_screen, 'MODE: MST', (5, 25),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))
else:
    cv2.putText(output_screen, 'MODE: None', (5, 25),
                cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255))

# gets key press
key = cv2.waitKey(1) & 0xFF

if DEBUG:
    cv2.imshow('debug window', gray)

cv2.imshow('graph window', output_screen)

# update the FPS counter
fps.update()


# gets key press
```

```
key = cv2.waitKey(1) & 0xFF

# Change program to CONNECT NODES mode
if key == ord('c'):
    if DEBUG:
        print('c hit')
    CONNECT_NODES_FLAG = True
    FIND_PATH_FLAG = False
    ADD_NODE_FLAG = False
    DRAW_PATH_FLAG = False
    FIND_MST_FLAG = False
    DELETE_NODE_FLAG = False
    NODE_CLICKED_COUNT = 0
    click_x = 0; click_y = 0

# Change program to FIND PATH mode
elif key == ord('f'):
    if DEBUG:
        print('f hit')
    FIND_PATH_FLAG = True
    CONNECT_NODES_FLAG = False
    ADD_NODE_FLAG = False
    DRAW_PATH_FLAG = False
    FIND_MST_FLAG = False
    DELETE_NODE_FLAG = False
    NODE_CLICKED_COUNT = 0
    click_x = 0; click_y = 0

# Change program to ADD NODE mode
elif key == ord('a'):
    if DEBUG:
        print('a hit')
    ADD_NODE_FLAG = True
    CONNECT_NODES_FLAG = False
    FIND_PATH_FLAG = False
    DRAW_PATH_FLAG = False
    FIND_MST_FLAG = False
    DELETE_NODE_FLAG = False
    NODE_CLICKED_COUNT = 0
    click_x = 0; click_y = 0

# Change program to DELETE NODE mode
elif key == ord('d'):
    if DEBUG:
        print('d hit')
    DELETE_NODE_FLAG = True
    ADD_NODE_FLAG = False
    CONNECT_NODES_FLAG = False
    FIND_PATH_FLAG = False
    FIND_MST_FLAG = False
```

```
            DRAW_PATH_FLAG = False
            NODE_CLICKED_COUNT = 0
            click_x = 0; click_y = 0

        # Change program to FIND MST mode
        elif key == ord('m'):
            if DEBUG:
                print('m hit')
            FIND_MST_FLAG = True
            DELETE_NODE_FLAG = False
            ADD_NODE_FLAG = False
            CONNECT_NODES_FLAG = False
            FIND_PATH_FLAG = False
            DRAW_PATH_FLAG = False

        # Change program to CALIBRATE MODE
        elif key == ord('k'):
            CALIBRATE = not CALIBRATE
            FIND_MST_FLAG = False
            DELETE_NODE_FLAG = False
            ADD_NODE_FLAG = False
            CONNECT_NODES_FLAG = False
            FIND_PATH_FLAG = False
            DRAW_PATH_FLAG = False

        # Change program to DEBUG mode
        elif key == ord('p'):
            if DEBUG:
                print('p hit')
            cv2.destroyWindow('debug window')
            DEBUG = not DEBUG

        # Prints current graph
        elif key == ord("g"):
            print('\n')
            print('GRAPH:')
            print('-'*25)
            for node in graph.list:
                print('node value: ', node.value)
                nbors = []
                for neighbor in node.neighbors:
                    nbors.append(neighbor.value)
                print('neighbors: ', nbors)
                print('-'*25)

        # break from the loop and exit program
        elif key == ord("q"):
            break

    # stop the timer and display FPS information
```

```
    fps.stop()
    print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
    print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

    # stops capture and closes opencv windows
    cap.release()
    cv2.destroyAllWindows()

    return


if __name__ == '__main__':
    main()
```

Program 2: google_text_detect.py

This program takes in a photo from a specified path and makes an API call to Google Cloud Visions API. It returns the text that is found in the image and displays it in the terminal. It also pipes the text found into an output file as well. This program is meant to read in handwritten code.

```
from google.cloud import vision
import io
from enum import Enum
from PIL import Image, ImageDraw
import os


img_path = '*Path to input image*'

output_prog = '*Path to output file*'


class FeatureType(Enum):
   PAGE = 1
   BLOCK = 2
   PARA = 3
   WORD = 4
   SYMBOL = 5


def draw_bounds(image, bounds, color):
   """Draw a border around the image using the hints in the vector list."""
   draw = ImageDraw.Draw(image)

   for bound in bounds:
       draw.polygon([
           bound.vertices[0].x, bound.vertices[0].y,
           bound.vertices[1].x, bound.vertices[1].y,
           bound.vertices[2].x, bound.vertices[2].y,
           bound.vertices[3].x, bound.vertices[3].y], None, color)
   return image


def get_image_info(image_file, feature):
   """Returns document bounds given an image."""
   client = vision.ImageAnnotatorClient()
```

```
bounds = []
output_str = ''

with io.open(image_file, 'rb') as image_file:
    content = image_file.read()

image = vision.types.Image(content=content)

response = client.document_text_detection(image=image)
document = response.full_text_annotation

# Collect specified feature bounds by enumerating all document features
for page in document.pages:
    for block in page.blocks:
        print('\nBlock confidence: {}\n'.format(block.confidence))
        for paragraph in block.paragraphs:
            print('    Paragraph confidence: {}'.format(
                paragraph.confidence))
            for word in paragraph.words:
                word_text = ''.join(
                    [symbol.text for symbol in word.symbols])
                print('        Word text: {} (confidence: {})'.format(
                    word_text, word.confidence))
                for symbol in word.symbols:
                    #print('\tSymbol: {} (confidence: {})'.format(
                        #symbol.text, symbol.confidence))
                    if (feature == FeatureType.SYMBOL):
                        bounds.append(symbol.bounding_box)
                    if symbol.property.detected_break.type == 1:
                        output_str += symbol.text + ' '
                    elif symbol.property.detected_break.type == 3:
                        output_str += symbol.text + '\t'
                    elif symbol.property.detected_break.type == 5:
                        output_str += symbol.text + '\n'
                    else:
                        output_str += symbol.text

                if (feature == FeatureType.WORD):
                    bounds.append(word.bounding_box)

            if (feature == FeatureType.PARA):
                bounds.append(paragraph.bounding_box)

        if (feature == FeatureType.BLOCK):
            bounds.append(block.bounding_box)

    if (feature == FeatureType.PAGE):
        bounds.append(block.bounding_box)
```

```
    print('-'*40)
    print('\nfinal output text:')
    print('\n'+output_str)
    print('-'*40)

    # The list `bounds` contains the coordinates of the bounding boxes.
    return bounds, output_str


# def render_image_text(filein, fileout):
def render_image_text(filein):
    image = Image.open(filein)
    bounds, out_str = get_image_info(filein, FeatureType.PAGE)
    draw_bounds(image, bounds, 'blue')
    bounds, out_str = get_image_info(filein, FeatureType.PARA)
    draw_bounds(image, bounds, 'red')
    bounds, out_str = get_image_info(filein, FeatureType.WORD)
    draw_bounds(image, bounds, 'yellow')
    image.show()

    F = open(output_prog, 'a+')
    F.write('\n'+out_str+'\n')
    F.close()

    os.system('code '+output_prog)


def main():

    render_image_text(img_path)

    return


if __name__ == '__main__':
    main()
```